Proyecto de Inteligencia Artificial

- Richard García de la Osa C512
- Carlos Alejandro Arrieta Montes de Oca C512
- Luis Alejandro Lara Rojas C512

Introducción

Transformada de Wavelet

La transformada de Wavelet es un tipo especial de transformada matemática que representa una señal en términos de versiones trasladadas y dilatadas de una onda finita (denominada wavelet madre).

En cuanto a sus aplicaciones, la transformada de Wavelet discreta se utiliza para la codificación de señales, mientras la continua se utiliza en el análisis de señales. Como consecuencia, la versión discreta de este tipo de transformada se utiliza fundamentalmente en ingeniería e informática, mientras que la continua se utiliza sobre todo en la física. Este tipo de transformadas están siendo cada vez más empleadas en un amplio campo de especialidades, a menudo sustituyendo a la transformada de Fourier, por su ventaja para el análisis de señales en el dominio del tiempo y la frecuencia.

Una de sus desventajas es que se debe seleccionar de antemano la wavelet a utilizar(wavelet madre). Este obstáculo frena a muchas personas de utilizar esta herramienta debido a que desconocen qué wavelet madre elegir, y generalmente la bibliografía sobre el tema es muy densa.

Wavelet ortogonales

Una wavelet ortogonal es una wavelet cuya transformada asociada es ortogonal . Es decir, la transformada wavelet inversa es el adjunto de la transformada wavelet.

Algoritmo de la Transformada Discreta de Wavelet

A la hora de desarrollar este algoritmo se tienen en cuenta wavelets discretas, y se redefine el problema en términos de dos filtros: - h: filtro de escalado - g: filtro de wavelet

Para computar la transformada se hallan unos coeficientes que son producto de hacer convoluciones a la señal con estos filtros.

El objetivo de este proyecto es desarrollar un modelo de Machine Learning que sea capaz de hallar los filtros h y g de una wavelet ortogonal dada una señal. Se pretende que estos filtros sean lo suficientemente buenos como para que se pierda la menor cantidad de información al aplicar la Transformada de Wavelet sobre la señal.

Idea general

El modelo desarrollado es una especie de auto-encoder, cuyos parámetros son los filtros h y g. El procedimiento es el siguiente: - Se aplica la Transformada de Wavelet sobre una señal de entrada X con los filtros h y g. - Al resultado se le aplica la Transformada inversa. Esto constituye la salida del modelo. - Mejorar los coeficientes de h y g por el descenso del gradiente teniendo en cuenta una función de pérdida(será definida más adelante). - Repetir el proceso desde el primer paso con los nuevos coeficientes de h y g para la señal X.

Este proceso termina cuando se cumplan ciertos criterios de parada.

Función de pérdida

Los filtros aprendidos por el modelo deben cumplir ciertas propiedades:

$\sum_k h[k]^2 = 1$

$\sum_k h[k] = \sqrt{2}$

$\sum_k g[k] = 0$

Los parámetros del modelo serán aprendidos por descenso del gradiente, por lo que se debe transformar estas restricciones en ecuaciones de penalidad diferenciables:

$(\sum_k h[k]^2 - 1)^2$

$(\sum_k h[k] - \sqrt{2})^2$

$(\sum_k g[k])^2$

Se define la función:

$L_w(h, g) = (\sum_k h[k]^2 - 1)^2 + (\sum_k h[k] - \sqrt{2})^2 + (\sum_k g[k])^2$

Nos interesa también que la señal de salidad x' del modelo sea lo suficientemente parecida a la entrada x. Para ello se agrega como penalidad el error cuadrático medio(mse) entre estas.

Función de pérdida:

$L = \alpha_1 mse(x, x') + \alpha_2 ||W||_1 + \alpha_3 L_w(h, g)$

En esta fórmula W es los coeficientes de la transformada de Wavelet aplicada a la señal x con los filtros h y g. α_1, α_2 y α_3 son hiperparámetros del modelo.

Detalles de Implementación

El código está implementado en python, y se encuentra en el archivo code.ipynb. Este archivo debe ejecutarse desde algún entorno que soporte jupyter notebook.

El primer bloque de código son las importaciones de las bibliotecas de python que vamos a utilizar:

```
import pywt
import numpy as np
from sklearn.metrics import mean_squared_error
import math
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

Luego se tienen las implementaciones de la transformada discreta de Wavelet y su inversa:

```
class dwt:
    def __init__(self, filter_size, h = None, g = None):
        if h is None:
            self.h = np.random.normal(1,2,filter_size)
        else:
            self.h = h

        if g is None:
            self.g = np.random.normal(1,2,filter_size)
        else:
            self.g = g

    def compute(self, input_):
        filter_bank = [self.h, self.g, np.flip(self.h), np.flip(self.g)]
        my_wavelet = pywt.Wavelet(name="my_wavelet", filter_bank=filter_bank)
        return pywt.dwt(input_, wavelet=my_wavelet)

    def inverse(self):
        return idwt(self)

    def update_weigths(self, hg, gg, lr = 0.001):
        self.h = self.h - lr*hg
        self.g = self.g - lr*gg

class idwt:
    def __init__(self, dwt):
        self.h = dwt.h
        self.g = dwt.g

    def compute(self, input_):
        ca,cd = input_
        filter_bank = [self.h, self.g, np.flip(self.h), np.flip(self.g)]
        my_wavelet = pywt.Wavelet(name="my_wavelet", filter_bank=filter_bank)
        return pywt.idwt(ca,cd, wavelet=my_wavelet)
```

La función de pérdida explicada anteriormente:

```
def loss_function(h, g, x, x_, W, reconstruction_weight = 1):
    l1 = 0.5
    l2 = 0.5
    r1 = reconstruction_weight
    s1 = mean_squared_error(x, x_)
    s2 = np.sum(np.abs(W))
    s3 = lhg(h,g)
    return r1*s1 + l1*s2 + l1*s3

def lhg(h, g):
    s1 = (np.sum(h**2) - 1)**2
    s2 = (np.sum(h) - math.sqrt(2))**2
    s3 = np.sum(g)**2
    return s1 + s2 + s3
```

Para el cálculo del vector gradiente resulta necesario calcular las derivadas parciales de la función respecto a cada una de las variables. Dada la dificultad del cálculo de la función derivada, se utiliza el cálculo de la variación de la función en un ϵ vecindad por cada una de las componentes del vector:

```
def compute_gradient_h(h, g, x, loss_function, diff = 0.001):
    length = len(h)
    gradient = np.zeros(length)
    mask = np.zeros(length)
    for i in range(length):
        mask[i] = diff
        h_plus = h + mask
        h_minus = h - mask

        dwtp = dwt(1, h_plus, g)
        idwtp = dwtp.inverse()

        dwtm = dwt(1, h_minus, g)
        idwtm = dwtm.inverse()

        Wp = dwtp.compute(x)
        x_p = idwtp.compute(Wp)

        Wm = dwtm.compute(x)
        x_m = idwtm.compute(Wm)

        mask[i] = 0

        lossp = loss_function(h_plus, g, x, x_p, Wp)
        lossm = loss_function(h_minus, g, x, x_m, Wm)
```

```
        gradient[i] = (lossp - lossm)/(2*diff)

    return gradient

def compute_gradient_g(h, g, x, loss_function, diff = 0.001):
    length = len(g)
    gradient = np.zeros(length)
    mask = np.zeros(length)
    for i in range(length):
        mask[i] = diff
        g_plus = g + mask
        g_minus = g - mask

        dwtp = dwt(1, h, g_plus)
        idwtp = dwtp.inverse()

        dwtm = dwt(1, h, g_minus)
        idwtm = dwtm.inverse()

        Wp = dwtp.compute(x)
        x_p = idwtp.compute(Wp)

        Wm = dwtm.compute(x)
        x_m = idwtm.compute(Wm)

        mask[i] = 0

        lossp = loss_function(h, g_plus, x, x_p, Wp)
        lossm = loss_function(h, g_minus, x, x_m, Wm)
        gradient[i] = (lossp - lossm)/(2*diff)

    return gradient

def compute_gradient(h, g, x, loss_function, diff = 0.001):
    hg = compute_gradient_h(h, g, x, loss_function, diff)
    gg = compute_gradient_g(h, g, x, loss_function, diff)
    return (hg, gg)
```

Luego se tiene la implementación del modelo:

```
class Model:
    def __init__(self,filter_size = 2**5):
        self.my_dwt = dwt(filter_size)
        self.my_idwt = self.my_dwt.inverse()
        self.losses = []
        self.min_loss = math.inf
```

```
def fit(self, x, epochs = 100, learning_rate = 0.001, verbose = True, good_error = None,
    best_h = self.my_dwt.h
    best_g = self.my_idwt.g
    for i in range(epochs):
        W = self.my_dwt.compute(x)
        x_ = self.my_idwt.compute(W)
        loss = loss_function(self.my_dwt.h, self.my_dwt.g, x, x_, W, reconstruction_weig
        self.losses.append(loss)
        if loss < self.min_loss:
            self.min_loss = loss
            best_h = self.my_dwt.h
            best_g = self.my_dwt.g

        if verbose:
            print('Epochs #' + str(i+1) + ": " + str(loss) + " loss")

        if not good_error is None and loss <= good_error:
            return

        hg, gg = compute_gradient(self.my_dwt.h, self.my_dwt.g, x, loss_function, diff)
        self.my_dwt.update_weigths(hg, gg, learning_rate)

    self.my_dwt = dwt(1, best_h, best_g)
    self.my_idwt = self.my_dwt.inverse()
    print("Best Loss", self.min_loss)

def predict(self, x):
    W = self.my_dwt.compute(x)
    return self.my_idwt.compute(W)

def wavelet(self):
    filter_bank = [self.my_dwt.h, self.my_dwt.g, np.flip(self.my_dwt.h), np.flip(self.my
    my_wavelet = pywt.Wavelet(name="my_wavelet", filter_bank=filter_bank)
    return my_wavelet

def dwt(self, x):
    ca, cd = self.my_dwt.compute(x)
    return np.concatenate([cd, ca])
```

La función fit de este modelo recibe los siguientes hiperpárametros, que constituyen las posibles condiciones de parada del algoritmo: - Iteraciones - Rating de aprendizaje - Límite de pérdida

Esta es la encargada de realizar las iteraciones del algoritmo expuesto anteriormente hasta cumplir una de las condiciones de parada.

A continuación la sección de Testing. Primero las funciones para generar coefi-

cientes de wavelet randomizados, y para generar una señal:

```
def generate_wave_coeff(length):
    result = []
    for _ in range(length*2):
        if np.random.random() < 0.9:
            result.append(0.0)
        else:
            result.append(np.random.uniform(-1,1))

    return (result[:length], result[length:])


def generate_signal(length, familie):
    ca, cd = generate_wave_coeff(length)
    x = pywt.idwt(ca, cd, familie)
    return x
```

Para generar la señal se recibe el tamaño de la señal, y la familia de funciones wavelet con que se va a generar.

Luego una función de distancia entre filtros:

```
def dist(f1, f2):
    max_l = max(len(f1), len(f2))
    min_l = min(len(f1), len(f2))
    diff = max_l - min_l
    best_i = 0

    if len(f1) == min_l:
        f1 = np.concatenate([f1, np.zeros(diff)])

    if len(f2) == min_l:
        f2 = np.concatenate([f2, np.zeros(diff)])

    distance = math.inf
    f1_norm2 = np.sqrt(np.sum(f1**2))
    f2_norm2 = np.sqrt(np.sum(f2**2))
    for i in range(max_l):
        current = 1 - (f1.dot(np.roll(f2,i))/(f1_norm2 * f2_norm2))
        if current < distance:
            distance = current
            best_i = i

    f2 = np.roll(f2, best_i)
    return distance
```

Con estas implementaciones se testeó el algoritmo implementado. Se selec-

cionaron distintas familias de funciones wavelet, por cada una de ellas se crearon varias señales aleatorias, y por cada señal además se generaron los tamaños de señal 2, 4, 6 y 8. Con cada combinación de estos parámetros se ejecutó el procedimiento, para luego guardar el promedio de los errores. Esto se explica con más detalles en la siguiente sección.

Resultados

Para probar el modelo se generaron coeficientes de wavelet aleatoriamente, y luego se le halló la transformada inversa con una wavelet fijada. El modelo debería recuperar wavelets similares a las wavelets usadas para generar los datos.

Se generaron 15 señales por cada wavelet. Para cada una de ellas se midió el Error de Reconstrucción, el Error de Wavelet Aprendida, el Error de la función de Escala aprendida y el Error de la Transformada Discreta de Wavelet con los coeficientes aprendidos.

- Error de Reconstrucción: Error cuadrático medio entre la señal x de entrada con la señal x' de salida.
- Error de Wavelet: Distancia entre la wavelet usada para generar la señal x y la wavelet aprendida. (La métrica usada para medir la distancia entre wavelets será definida más adelante)
- Error de Escala: Distancia entre la función de Escala aprendida y la usada para generar la señal x.
- Error de DWT: Error cuadrático medio entre la DWT con la wavelet usada para generar x y la wavelet aprendida por el modelo.

Distancia entre wavelets:

$dist(h1, h2) = min_{0<=i<k} 1 - (\frac{dot(h1, shift(h2, i))}{||h1||_2 ||h2||_2})$

dot: producto escalar de los vectores.

shift(h2, i): corrimiento circular de h2 i posiciones a la derecha.

k: max(len(h1), len(h2))

El filtro de menor tamaño entre h1 y h2 debe ser rellenado con ceros hasta alcanzar el tamaño del otro filtro.

Para el experimento se aplicaron 4 modelos para cada señal, con tamaños de filtro 2, 4, 6 y 8 respectivamente. Se seleccionó el resultado del modelo que minimizó el error de reconstrucción.

Hiperparámetros seleccionados: - 2000 iteraciones - 0.001 rating de aprendizaje - 4 límite de pérdida(Se para de entrenar el modelo cuando el valor de la pérdida era menor a este) - $\alpha_1 = 10$, $\alpha_2 = 0.5$, $\alpha_3 = 0.5$

La tabla siguiente muestra los datos obtenidos:

	Wavelets	Reconstruction Error	Wavelet Error	Scaling Error	DWT Error
0	haar	0.0417028	0.433294	0.201882	0.0379189
1	db1	0.214974	0.0598129	0.107658	0.00432762
2	db2	0.0799952	0.331267	0.315995	0.0785023
3	db3	0.0269186	1.3823e-06	0.122895	0.0492721
4	sym2	0.0568944	0.309908	0.179937	0.0137322
5	sym3	0.0677578	0.516305	0.46486	0.0126608
6	sym4	0.00884499	0.309241	0.0546851	0.072664
7	coif1	0.0302337	0.313961	0.0689839	0.0495169

En la columna Wavelet se muestran nombres de wavelets correspondientes a la biblioteca de python *PyWavelets*. Los datos mostrados en cada celda corresponde al promedio de error de las 15 señales generadas en cada caso.

Conclusiones

Teniendo en cuenta los resultados obtenidos anteriormente, se puede apreciar que el algoritmo tiene gran efectividad ya que los errores promedio de la transformada de wavelet con los filtros aprendidos con respecto a los filtros prefefinidos se han mantenido todos por debajo de 0.1, lo cual es un indicador de la alta precisión de esta solución.

Sin embargo, al ejecutar este algoritmo con un numéro de iteraciones no muy grande(2000 iteraciones) por cada señal, se puede apreciar una demora considerable si se pretende obtener eficiencia para realizar los cálculos de wavelet.

Por ello se pudiera hacer uso de algunas bibliotecas de python que exploten mejor la GPU en aras de aumentar la velocidad de ejecución. Se deja como recomendación realizar una investigación acerca de qué bibliotecas podrían usarse con este propósito.